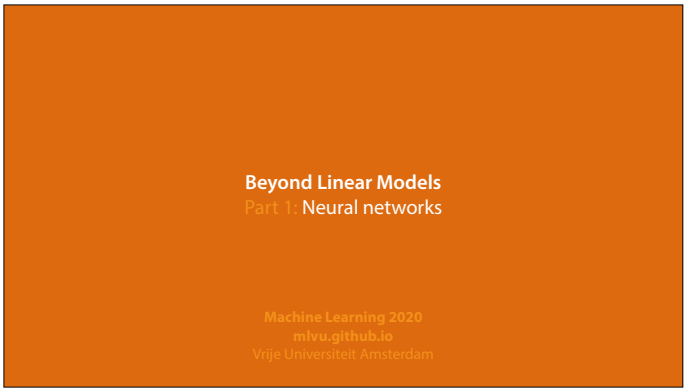
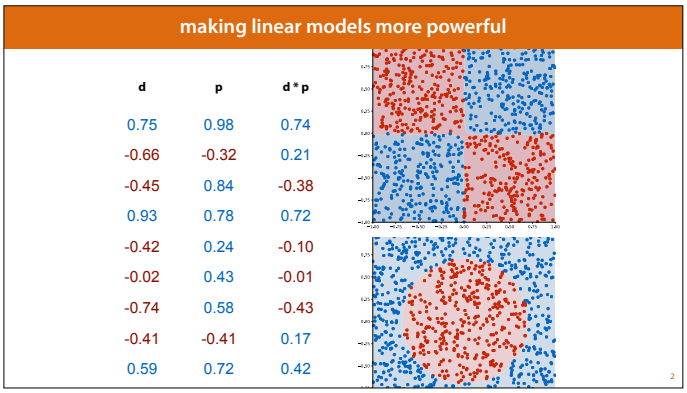
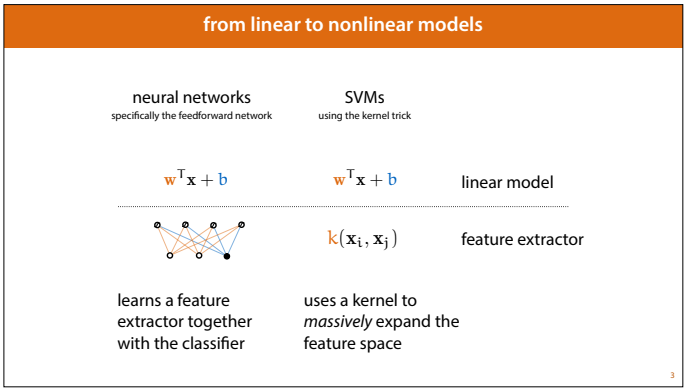


|section|Neural networks|
|video|https://www.youtube.com/embed/DeQ4STHYT3g|



A few lectures ago, we saw how we could make a linear model more powerful, and able to learn nonlinear decision boundaries by just **expanding our features**: we add new features derived from the old ones, and depending on which combinations we add, we can learn new, non-linear decision boundaries or regression functions.



Both models we will see today, **neural networks** and **support vector machines**, take this idea and build on it. Neural networks are a big family, but the simplest type, the **two-layer feedforward network**, functions as a feature extractor followed by a linear model. In this case, we don't choose the extended features but we *learn* them, together with the weights of the linear model.

The support vector machine doesn't learn the expanded features (we still have to choose them manually), but it uses a **kernel function** to allow us to fit a linear model in a *very* high-dimensional feature without having to pay for actually computing all these expanded features.

Using the SVM in this way is becoming less popular. Their loss function (the maximum margin loss) is still used, but the feature expansion trick is exceedingly rare. For this reason, we have made this part of the lecture optional*. We suggest you have at least a quick look at the basic idea, but the material covered in the fourth part of this lecture won't appear on the exam.*

linear models 2

1985–1995
part 1: Feedforward Neural Networks

part 2: Scalar Backpropagation

1995–2005
part 3: Support Vector Machines, Hinge loss

part 4: Optimizing under constraints
Lagrange optimization

The kernel Trick

4

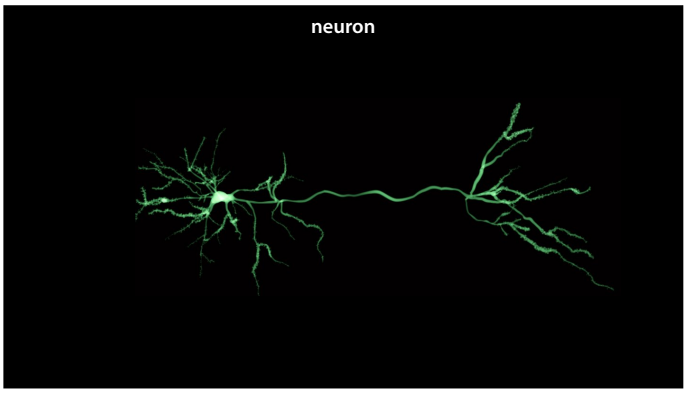
The layout of today's lecture will be largely chronological. We will focus on **neural networks**, which were very popular in the late eighties and early nineties.

Then, towards the end of the nineties, interest in neural networks died down a little and **support vector machines** became much more popular.

In the next lecture, we'll focus on Deep Learning, which sees neural networks make a comeback in a big way.

---

neuron



In this video, we'll start with the basics of neural networks.

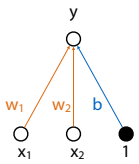
In the very early days of AI (the late 1950s), researchers decided to try a simple approach: the brain is the only truly intelligent system we know, so let's see what it's made of, and whether that provides some inspiration for intelligent (and learning) computer systems.

They started with a single brain cell: a neuron. A neuron receives multiple different input signals from other cells through connections called **dendrites**. It processes these in a relatively simple way, deriving a single new signal, which it sends out through its single **axon**. The axon branches out so that this single output signal can reach different cells.

image source: http://www.sciencealert.com/scientists-build-an-artificial-neuron-that-fully-mimics-a-human-brain-cell

---

1957: the perceptron



$y = w_1x_1 + w_2x_2 + b$

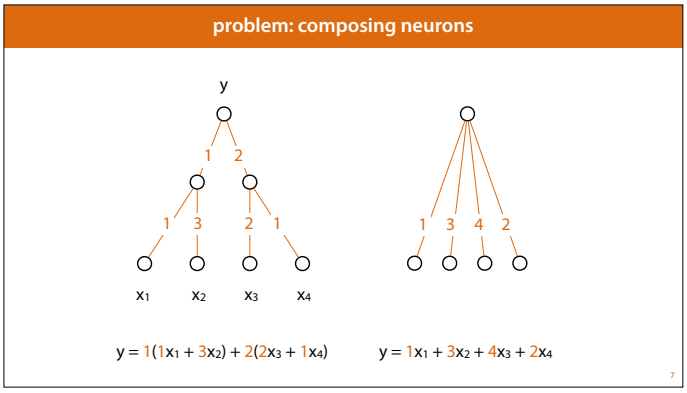
output pos if $y > 0$
output neg otherwise

These ideas needed to be radically simplified to work with computers of that age, but the basic idea was still there: multiple inputs, one output. Doing this yielded one of the first successful machine learning systems: the **perceptron**. This was the model we saw in action in the video in the the first lecture.

The perceptron has a number of inputs, the *features* in modern parlance, each of which is multiplied by a weight. The result is summed, together with a bias parameter, and the sign of this result is used as the classification.

Of course, we've seen this classifier already: it's just our basic linear classifier. The training algorithm was a little different from gradient descent, but the basic principle was the same.

Note that when we draw the perceptron this way, as a mini network, the bias can be represented as just another input that we fix to always be 1. This is called a **bias node**.

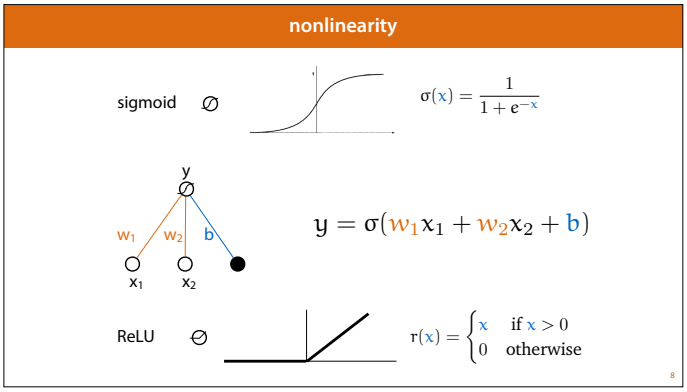


Of course the brain's power does not come from the fact that a single neuron is such a powerful mechanism by itself: it's the *composition* of many simple parts that allows it to do what it does. We make the output of one neuron the input of another, and build networks of billions of neurons.

And this is where the perceptron turns out to be too simple an abstraction. Because **composing perceptrons doesn't make them more powerful**. Consider the graph on the left, with multiple perceptrons composed together.

Writing down the function that this graph represents, we see that we get a simple function, with the first two perceptrons in brackets. If we then multiply out the brackets, we see that the result is a linear function. This means that we can represent this function also as a single perceptron with four inputs. This is always true. No matter how many perceptrons you chain together, the result will never be anything more than a simple linear function over your inputs: a single perceptron.

We've removed the bias node here for simplicity, but the conclusion is the same with a bias node included.



To create perceptrons that we can chain together in such a way that the result will be more expressive than any single perceptron could be, the simplest trick is to include a **non-linearity**, also called an **activation function**.

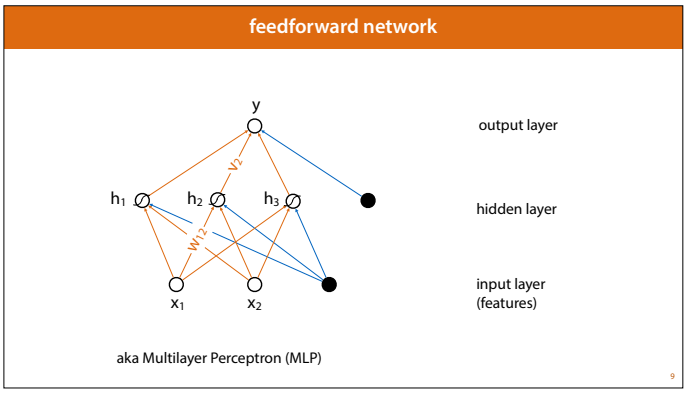
After all the weighted inputs have been combined, we pass the resulting scalar through a simple non-linear scalar function to produce the output. One popular option, especially in the early days of neural networks, was the **logistic sigmoid**, which we've seen already. Applying a sigmoid means that the sum of the inputs can range from negative infinity to positive infinity, but the output is always in the interval [0, 1].

Another, more recent non-linearity is the linear rectifier, or **ReLU** nonlinearity. This function just sets every negative input to zero, and keeps everything else the same.

Not using an activation function is also called using a **linear activation**.

It's difficult to provide an intuition here for quite how these

nonlinearities operate in the larger model. For now, the only point we want to make is that adding nonlinearities stops a network of perceptrons from collapsing into a single nonlinear function. We'll trust that these functions allow the network to learn some useful functions. We'll see a little more intuition in later lectures.



Using these nonlinearities, we can arrange single perceptrons into **neural networks**. Any arrangement of perceptrons makes a neural network, but for ease of training, this arrangement seen here was the most popular for a long time. It's called the **feedforward network** or **multilayer perceptron (MLP)**. We arrange a layer of **hidden units** in the middle, each of which acts as a perceptron with a nonlinearity, connecting to all input nodes. Then we have one or more output nodes, connecting to all hidden units. Note the following points.

- There are no cycles, the network feeds forward from input to output.
- Nodes in the same layer are not connected to each other, or to any other layer than the next and the previous one.
- Each layer is **fully connected** to the previous layer, every node in one layer connects to every node in the layer before it.

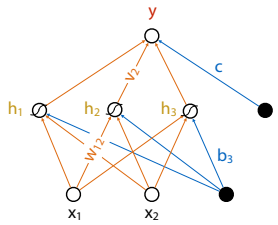
In the 80s and 90s these networks usually had just one hidden layer, because we hadn't figured out how to train deeper networks.

Nowadays it's common to find feedforward networks with as many as 12 hidden layers, often used as part of a much larger network also employing different types of layers.

Note that every line in this picture represents one distinct parameter of the model. The blue lines (those connected to bias nodes) represent biases, and the rest represent weights.

We can use networks like these to do classification or regression.

in pseudocode




```
for j in 1 … 3:
    for i in 1 … 2:
        kⱼ += wᵢⱼ * xᵢ
    kⱼ += bⱼ

for i in 1 … 3:
     hᵢ = sigmoid(kᵢ)

for i in 1 … 3:
     y += hᵢ * vᵢ
y += c

loss = (y - t) ** 2
```

10

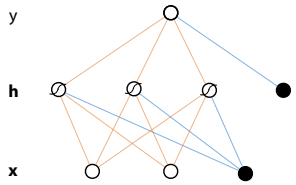
This is a simple feedforward neural network in pseudocode (including the computation of the loss). It's worth studying this in detail to make sure you understand all the steps.

Here are some questions you can ask yourself to see if you properly understand.

- Where is k_2 in the diagram?
- How many values b_j are there, and to which parts of the diagram do they correspond?
- What loss function are we using here?
- Why does the first layer have two nested loops, but the second only one?
- The computation of the sigmoid activation happens only in one loop. If the network looked different, would that happen in two nested loops, like the computation of the first layer?

---

regression



linear regression

feature expansion

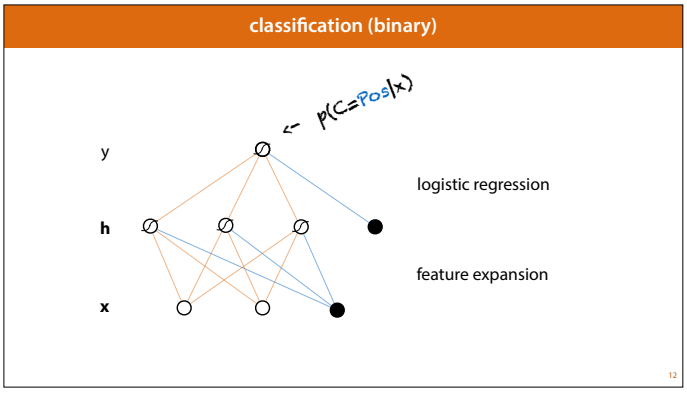
11

To build a **regression model**, all we need is one output node without an activation. This means that our network as a whole, describes a function from the feature space to the real number line.

We can think of the first layer of our network as computing a *feature expansion*: the same thing we did in the fourth lecture to enable our linear regression to learn non-linear patterns, but this time, we don't have to come up with the feature expansion ourselves, we simply learn it. The second layer is then just a linear regression in this expanded feature space.

The number of hidden nodes is a hyperparameter. More nodes makes the network more powerful (that is, able to represent more different functions), but also more likely to overfit, more expensive to compute and potentially more difficult to train. The only real advice we can give is that whenever possible, your hidden layer should be wider than the input layer.

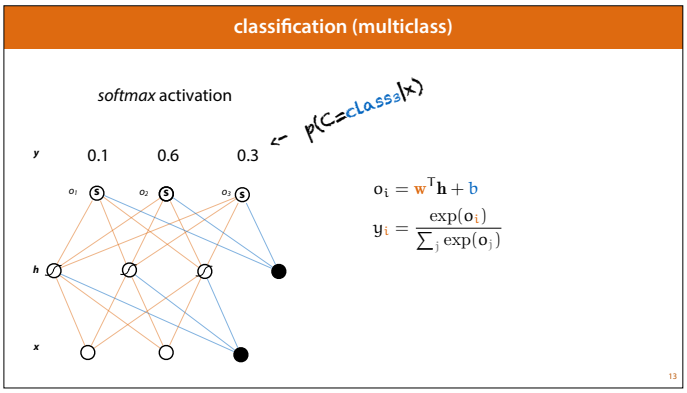
After we've computed the output, we can apply any regression loss function we like, such as least-squares loss.



To build a **binary classifier**, we could do what the perceptron did: use the *sign* of the output as the class. This would be a bit like using our least squares classifier from the second lecture, except with a feature expansion layer below it.

These days, however, it's much more common to take inspiration from the *logistic regression*. We apply the logistic sigmoid to the output and interpret the resulting value as the probability that the given input (x) is of the positive class.

The logarithmic loss that we use for logistic regression, can then be applied here as well.



For multiclass classification, we can use something called a **softmax activation**. We create a single output node for each class, and then ensure that **they are all positive and that together they sum to one**. This allows us to interpret them as **class probabilities**.

The softmax function is one way to ensure this property. It simply passes each output node through the exponential function, to ensure that they are all positive, and then divides each by the sum total, to ensure that all outputs together sum to one.

After the softmax we can interpret the value of node y_3 as the probability that **x** has class 3. Given these probabilities, we can apply a simple log loss: the aim is to maximize the logarithm of the probability of the true class.

The softmax is slightly unusual among activations in that it exchanges information between the nodes of the output layer. This allows us to look at just one output, and still provide a learning signal for all output nodes. For instance, imagine that the probability of class3 is low, but it should be high. This means that the value of y_3 should increase, but because the three nodes are required to sum to one, it automatically means that the values of y_1 and y_2 should decrease. We will see in the next video how this is achieved.

training: *stochastic* gradient descent

pick random weights **w** (for the whole model)

loop:

  for x in X:

    $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \text{loss}_{\mathbf{x}}(\mathbf{w})$

14

Because neural networks can be expensive to compute we tend to use ***stochastic* gradient descent** to train them.

Stochastic gradient descent is very similar to the gradient descent we've seen already, but we define the loss function over a **single example** instead of summing over the whole dataset: just use the same loss function, but pretend your data set consists of only one instance. We then loop over all instances, and perform a small gradient descent step for each one based on only the loss for that instance.

Stochastic gradient descent has many advantages, including:

- Using a new instance each time adds some noise to the process, since the gradient will be slightly different for each instance, which can help to escape local minima.
- Gradient descent works fine if the gradient is not perfect, but still good on average (over many instances). This means that taking many small inaccurate steps is often much better than taking one very accurate big step.
- Computing the loss over the whole dataset is expensive. By computing the loss over one instance at a time, we get N steps of stochastic gradient descent for the price of one step of regular gradient descent.

*The most common approach these days is a compromise between stochastic and regular gradient descent, where we actually compute the loss for a small **batch** of instances (say 32 of them), and take a single step of gradient descent for each batch. This is called **minibatch gradient descent,** which we'll look at more closely next lecture.*

---

summary: training a neural network

get some examples of input and output

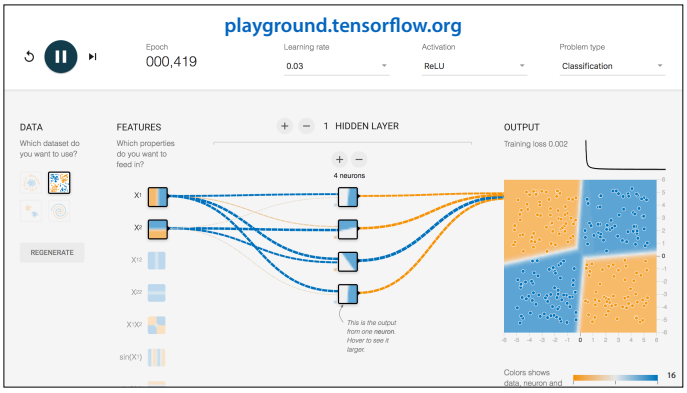
get a loss function
least squares, cross entropy

work out the gradient of the loss wrt the weights

use (stochastic) gradient descent to improve the weights
bit by bit.

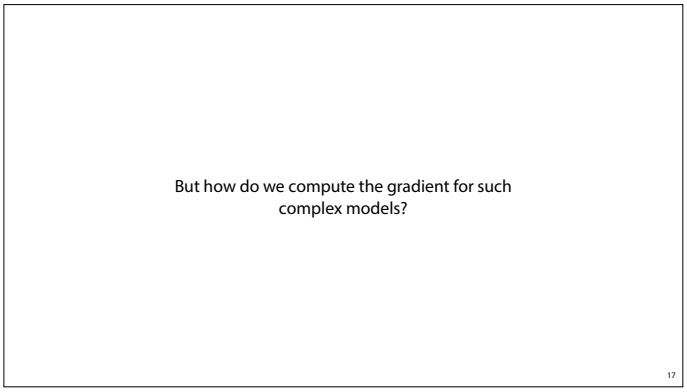
15

Apart from this exception, the training of a neural network proceeds in much the same way as the training of linear classifiers we've seen already.



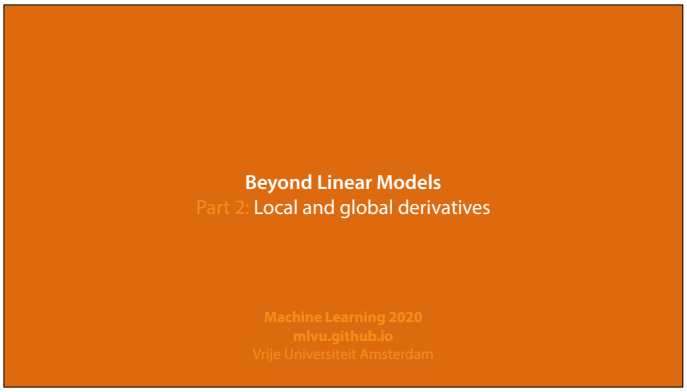
Before we dig into the details, we can get a sense of what neural network training looks like in the **tensorflow playground**. We suggest you play around a bit with the different datasets, different activations, and try to change the shape of the network.

- Note how the shape of the decision boundary changes based on the activation functions we choose (curvy for sigmoid, piecewise linear for ReLU)
- Note that adding another layer makes the network much more difficult to train (especially with sigmoid activations).
- Try the linear activation (i.e. no activations on the hidden nodes). Note that all you get is a linear decision boundary, not matter how many layers you try.
- Try a network on the circular dataset, with hidden layers with 2 units. It should not be possible so solve the circular dataset this way. It can be shown that to create a closed shape like a circle as a decision boundary, at least one hidden layer needs to be strictly bigger than your input layer.

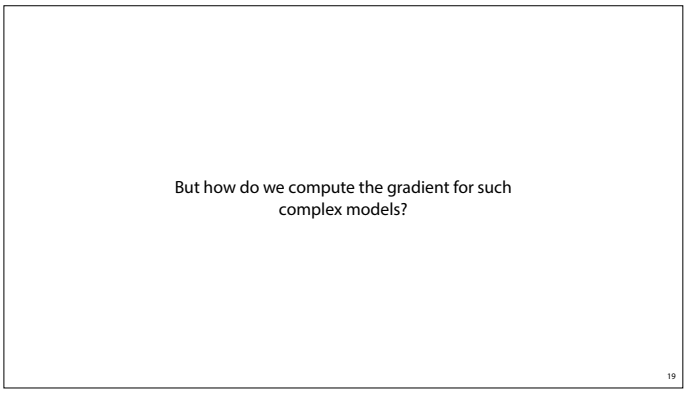


That's the basic idea of neural networks. So far, it's hopefully a pretty simple idea. The complexity of neural networks lies in computing the gradients. For such complex models, sitting down at the kitchen table with pen and paper, and working out a symbolic expression for the gradient is no longer feasible. If we manage it at all, we get horrible convoluted expressions that no longer reduce to nice, simple functions, as they did in the case of linear regression and logistic regression.

To help us out, we need the **backpropagation** algorithm, which we'll discuss in the next video.

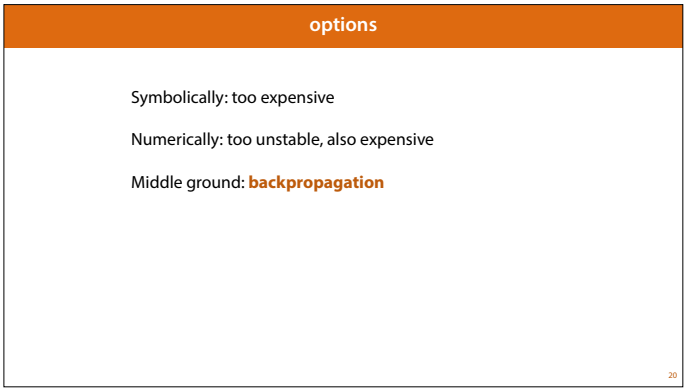


|section|Local and global derivatives|
|video|https://www.youtube.com/embed/qERQttAL-nE|



In the last video, we saw what the structure of a very basic neural network was, and we ended on this question. **How do we work out the gradient?**

For neural networks, the gradients quickly get too complex to work out by hand, so we need to automate this process.



There are three basic flavors of working out derivatives and gradients automatically.

The first is to do it **symbolically**, What we do on pen and paper, when we work out a derivative, is a pretty mechanical process. It's not too difficult to program this process out and let the computer do it for us. This, is what happens, when you ask Wolfram alpha to work out a derivative, for instance. It has its uses, cartainly, but it won;t work for us. The symbolic expression of the gradient of a function grows exponentiall with the complexity of the original function. That means that as we build bigger and bigger networks the expression of the gradient would soon grow too big to store in memory, let alone to compute.

An alternative approach is to forget the symbolic form of the function, and just **estimate the gradien**t for a specific input **x**. We could, for instance, pick some points close to **x** and and fit a hyperplane through the outputs. This would be a pretty good approximation of the tangent hyperplane, so we could just read out the gradient. The problem is that this

is a pretty unstable business. It's quite difficult to be sure that the answer is accurate. It's also expensive: the more dimensions in your model space, the more points you need to get an accurate estimate of your gradient, and each point requires you to recompute your model for a new input.

Backpropagation is a middle ground: it does part of the work symbolically, and part of the work numerically. We get a very accurate computation of the gradient, and the cost of computation is usually only twice as expensive as computing the output for one input.

---

backpropagation

Break your computation down into a chain of *modules*.

Work out the derivative of each module with respect to its input symbolically.

Compute the global gradient *for a given input* by multiplying these gradients.

Accumulate your gradients down the computation graph. next part

21

Here are the three steps required to implement backpropagation for a given function.

Don't worry if this seems abstract, it should become clearer when we look at an example.

We'll focus in the first three steps in this part. In the next part, we'll show how to build on this for complex computation graphs.

---

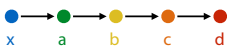
example

$$f(x) = \frac{2}{\sin(e^{-x})} \qquad f(x) = d(c(b(a(x))))$$

modules:

$$d(c) = \frac{2}{c}$$
$$c(b) = \sin b$$
$$b(a) = e^a$$
$$a(x) = -x$$

computation graph:



22

To show that backpropagation is a *generic* algorithm for working out gradients, not just a method for neural networks, we'll first show how it works for some arbitrary scalar function: $f(x) = 2/\sin(e^{-x})$.

There is no special meaning to this function. I just chained together a few operations for which the derivatives are simple.

First we take our function f, and we break it up into a chain of smaller functions, the output of each feeding into the next. Defining the functions a, b, c, and d as shown, we can write $f(x) = d(c(b(a(x))))$.

The graph on the right is a called a **computation graph**: each node represents a small computer program that receives an input, computes an output and passes it on to another module.

Normally, we wouldn't break a function up in such small modules: this is just a simple example to illustrate the principle.

chain rule

$$\frac{\partial f(x)}{\partial x} = \frac{\partial d(c(b(a(x))))}{\partial c(b(a(x)))} \frac{\partial c(b(a(x)))}{\partial x} = \frac{\partial d}{\partial c} \frac{\partial c}{\partial x}$$

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial x}$$

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial c} \frac{\partial c}{\partial x}$$

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial x}$$

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial x}$$

$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial b} \frac{\partial b}{\partial x}$$

$$\frac{\partial b}{\partial x} = \frac{\partial b}{\partial a} \frac{\partial a}{\partial x}$$

Because we've described our function as a composition of modules, we can work out the derivative purely by repeatedly applying the chain rule.

Since we know for each function what the argument is, we'll leave the arguments out to keep the notation clean.

example

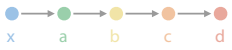
$$f(x) = \frac{2}{\sin(e^{-x})}$$

$$d(c) = \frac{2}{c}$$

$$c(b) = \sin b$$

$$b(a) = e^a$$

$$a(x) = -x$$



local derivatives

$$\boxed{\frac{\partial f}{\partial x}} = \boxed{\frac{\partial d}{\partial c}} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial x}$$

global derivative

We'll call the derivative of the whole function with respect to input x the **global derivative**, and the derivative of each module with respect to its input we will call a **local derivative**.

work out *local* derivatives

symbolically

$$d(c) = \frac{2}{c}$$

$$c(b) = \sin b$$

$$b(a) = e^a$$

$$a(x) = -x$$

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial x}$$

$$\frac{\partial f}{\partial x} = -\frac{2}{c^2} \cdot \cos b \cdot e^a \cdot -1$$

The next step is to work out the local derivatives symbolically, using the rules we know.

The difference from what we normally do is that **we stop** when we have the derivatives of the output of a module in terms of the input. For instance, the derivative $\partial c / \partial b$ is cos b. Normally, we would fill in the definition of b and see if we could simplify any further. Here we stop once we know the derivative in terms of b.

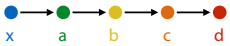
compute a forward pass (x=−4.499)

retain values of a, b, c, d

$$f(-4.499) = 2$$

$$d = \frac{2}{c} = 2$$
$$c = \sin b = 1$$
$$b = e^{a} = 90$$
$$a = -x = 4.499$$

computation graph:



26

Then, once all the local derivatives are known, in symbolic form, we switch to **numeric computation**. We will take a *specific* input, in this case -4.499 and compute the gradient only for that.

First we compute the output of the function f given this input. We do this simply by following the computation graph: the input is fed to the first module, and its output is fed to the second module, and so on. This is known as the **forward pass**. During our computation, we also retain our intermediate values a, b, c and d. These will be useful later on.

---

compute the backward pass

numerically

$$f(-4.499) = 2$$
$$d = \frac{2}{c} = 2$$
$$c = \sin b = 1$$
$$b = e^{a} = 90$$
$$a = -x = 4.499$$

$$\frac{\partial f}{\partial x} = -\frac{2}{c^2} \cdot \cos b \cdot e^{a} \cdot -1$$
$$= -\frac{2}{1^2} \cdot \cos 90 \cdot e^{4.499} \cdot -1$$
$$= -\frac{1}{2} \cdot 0 \cdot 90 \cdot -1 = 0$$

27

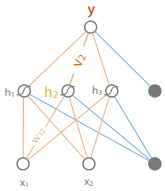
Next up is the **backward pass**. We take the chain-rule derived form of the derivative, and we fill in the intermediate values a, b, c and d.

This gives us a function with no variables, so we can compute the output. The result is that the derivative of this function, for the specific input -4.499, is 0.

Note that we have stopped doing symbolic computations: we fill in the numeric values and work out the numeric result (accepting a small amount of inaccuracy due to floating point imprecisions).

---

t: training data



$$l = (y - t)^2$$
$$y = v_1 h_1 + v_2 h_2 + v_3 h_3 + b_h$$

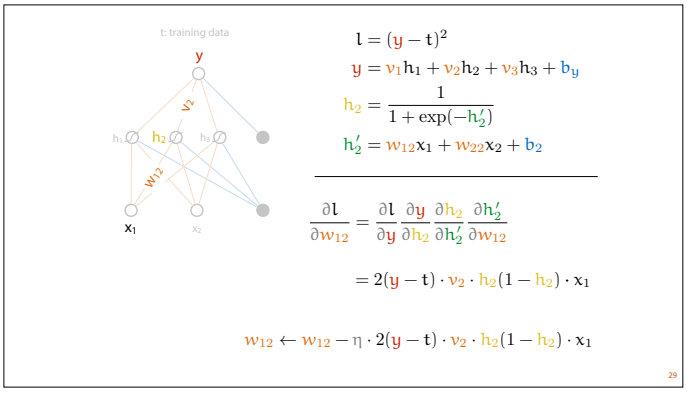
$$\frac{\partial l}{\partial v_2} = \frac{\partial l}{\partial y} \frac{\partial y}{\partial v_2}$$
$$= 2(y - t) \cdot h_2$$

$$v_2 \leftarrow v_2 - \eta \cdot 2(y - t) \cdot h_2$$

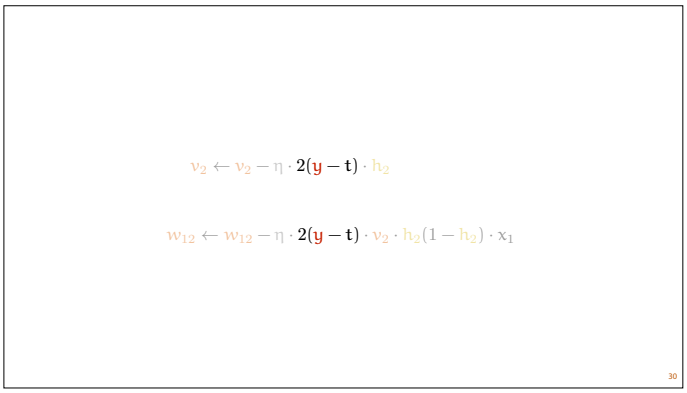
28

Here's what the local gradients look like for the weight v_2.

The line on the bottom shows how we update v_2 when we apply a single step of stochastic gradient descent for x (x may not appear in the gradient, but the values y and h2 were computed using x).

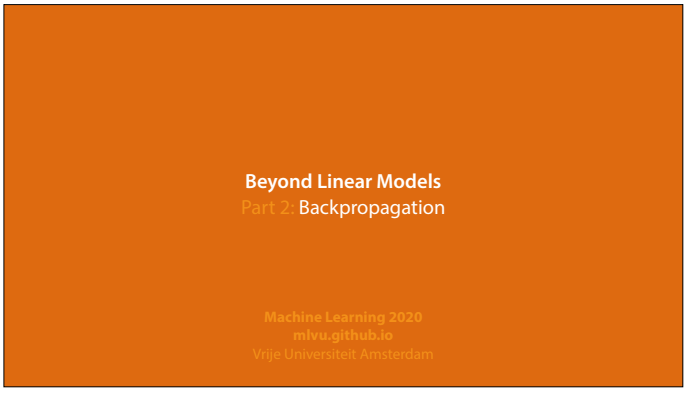


So far, this is no different from gradient descent on a linear model. The real power of the backpropagation algorithm shows when we look at how the error propagates back down the network (hence the name) and is used to update the weights. Lets look at the derivative for weight w12



This approach by itself gives us a way to work out all the derivatives we need. But note that we are recomputing the same quantity multiple times. For instance, the error term 2(y-t) appears in every update rule we compute.

This is no coincidence. Because of the graph structure of our computation, the same local derivatives will show up in the expressions for many of our global derivatives. We can make clever use of this to compute all gradients efficiently. We will add this ingredient in the next part of the lecture.



|section|Backpropagation|
|video|https://www.youtube.com/embed/TwHVP7FAT6I|

backpropagation

Break your computation down into a chain of modules.

Work out the derivative of each module with respect to its input symbolically.

Compute the global gradient for a given input by multiplying these gradients.

previous part

Accumulate your gradients down the computation graph.

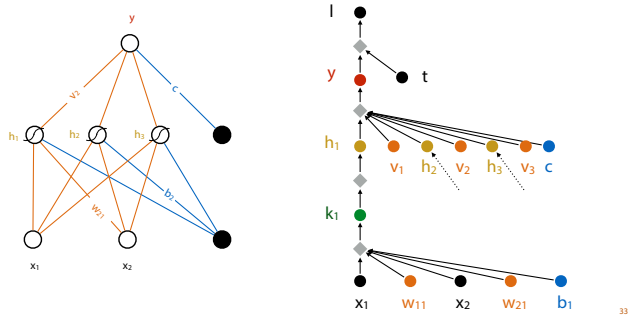
32

This is how we described backpropagation in the last part. There, we focused on making a tradeoff between numeric and symbolic computation, and working out local derivatives.

However, there is another ingredient to backpropagation, which will show us where the name comes from. If we carefully *accumulate* our computed gradients we will see that we can compute all derivatives we need in a single walk down the graph.

---

a computation graph



33

The first thing we have to do is draw a proper **computation graph**. The diagram we've drawn so far provides a kind of "model perspective": it separates the *inputs and intermediate* values, which are on the nodes, from the *parameters*, which are on the edges.

In a proper computation graph, all the values that go into our computation and come out of it, are nodes, whether they're parameters of a model or inputs. We'll draw these as circles. To represent the *computations*, we'll introduce a new type of node, drawn as a diamond: ◆. The edges tell us which values are the input of a particular computation, and which is the output. All edges are directed, the ones going in to the computation represent inputs, and the ones coming out represent outputs.

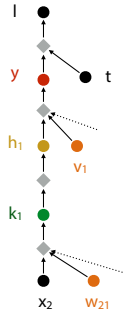
In a computation graph, a circle is always connected to a diamond and vice versa.

We've drawn only part of the computation graph for this network here, to keep things simple. (The full computation graph for this network would have 22 nodes.)

Note also that:

- We've included **the computation of the loss**. This is because when we compute gradients, we're always interested in the derivative of the loss with respect to the parameters. Therefore, the computation of the loss should be part of the computation graph.
- We've separated the computation of the unactivated hidden nodes (k) and the activated ones (h). We could make this one computation, but it's more common to separate them.

a computation graph



$$l = (y - t)^2$$

$$y = v_1 h_1 + v_2 h_2 + v_3 h_3 + c$$

$$h_1 = \frac{1}{1 + \exp(-k_1)}$$

$$k_1 = w_{11} x_1 + w_{21} x_2 + b_1$$

34

Here are the computations represented by the diamond nodes.

We've removed some further parts of the graph for the sake of clarity.

backpropagation

Do a forward pass.
Compute the output (loss) given the inputs.

Start at the top (output) of your computation graph.
The computation graph should always have a *single* output.

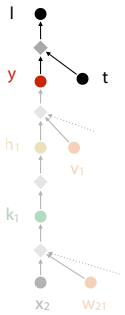
Compute the derivative for *every* node
That is, the derivative of the output wrt. the node.

Work your way down.
As we will see, this makes the computation entirely *local*.

35

For a complex computation graph, it's important to work the derivatives out in the right order. This allows us to *reuse* what we've already computed at every step.

The algorithm is simple. We start at the top and work our way down.



$$l = (y - t)^2$$

$$\frac{\partial l}{\partial y} = 2(y - t) \qquad \frac{\partial l}{\partial t} = -2(y - t)$$

imaginary gradient descent update

$$y \leftarrow y - \eta \frac{\partial l}{\partial y} = y - 2\eta(y - t)$$

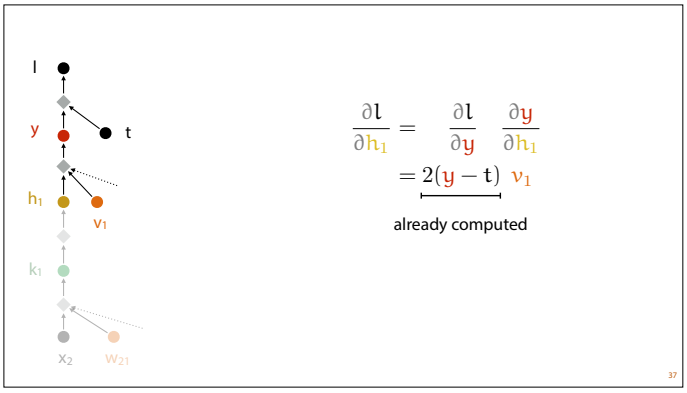
36

We start at the topmost computation node. We focus on the *computation* of the loss from the prediction y and the target t. We'll compute derivatives for every node in our computation graph.

These derivatives are not directly useful to us yet, because neither y nor t are values we can change directly: t, the target, is given by the data so we can't change that at all, and y we can only change indirectly by changing our parameters.

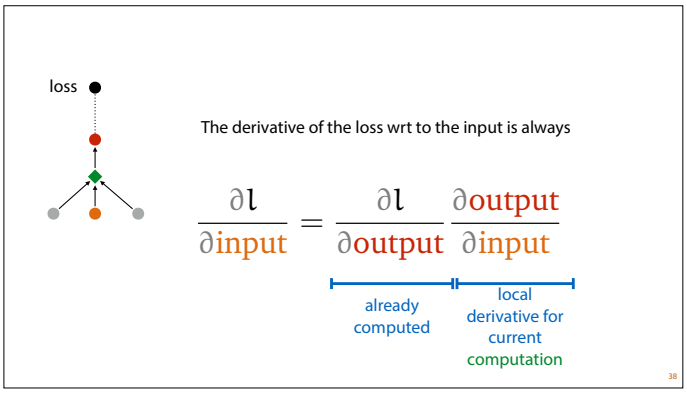
What we can do however, is *imagine* that we could change y directly. In that case this derivative tells us how we would update y. In other words, this derivative tell us how we would *like* to change y, even though we can't.

Note how the sign is taken into account. If t is larger than y, the error term y - t is negative and the gradient update tells us to add a little bit to y. Likewise if t is smaller than y, we end up subtracting from y. We can't do this directly, but the derivative tells us what we would like to achieve.



Next, we move to the previous computation, the one that takes h_1, v_1 and c as input, and produces y. We can compute a derivative for each input,]] but let's focus on h_1.

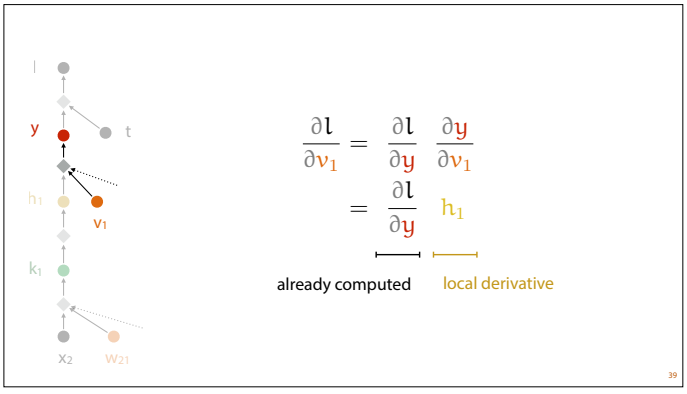
Applying the chain rule tells us that we can break up the derivative of the loss wrt to h_2 into the derivative of the loss with respect to y times the derivative of y with respect to h_1.



This shows us a **general rule** about backpropagation on computation graphs. If we have a node feeding into a computation, the global derivative of that node (the loss over the value of the node) is always the global derivative of the output of the computation, times the local derivative of the output over the input.

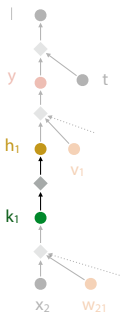
The key to backpropagation is that if we traverse the graph in the right order, the first derivative (the loss over the output) is always something *we have already computed*. If we take care that work out the derivatives in the reverse order in which we computed them, we will only ever need to work out the local derivative of the current computation.

In essence, we are applying the chain rule only once for each computation. We are breaking the computation if the loss from this input into the computation that we are focusing on, and the rest of the computation graph that follows afterward.



With this notion in hand, we can work our way down the graph. For every input of every computation, we can work out the derivative in terms of the derivative for the output. We don't need to worry about what happens *after* the output: the derivative for the output is all we need.

Once we have the local derivative worked out, we fill in the values from the forward pass, and multiply them by the value we already have for the derivative for the output.

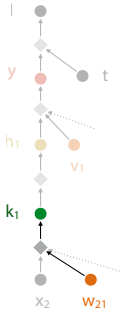


$$\frac{\partial l}{\partial k_1} = \frac{\partial l}{\partial h_1}\frac{\partial h_1}{\partial k_1}$$
$$= \frac{\partial l}{\partial h_1}\sigma(k_1)(1-\sigma(k_1))$$
$$= \frac{\partial l}{\partial h_1}h_1(1-h_1)$$

40

Once we are done with a computation, we just move down the computation graph to the next one. We don't care whether nodes represent parameters of the model like v_1 (which we can change directly) or intermediate values like h_1 (which we can only change indirectly). We want derivatives for all nodes in the graph, because we'll need them to work out derivatives for any node below.

In this case, we deal with the node that computes the activation function. Since we already have the derivative of the loss with respect to h_1, we only need to apply the chain rule *once*. This breaks up the computation of the derivative that we need (l over k_1) into one derivative we've already computed (l over h_1) and one local derivative that we can easily work out (h_1 over k_1).



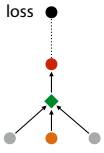
$$\frac{\partial l}{\partial w_{21}} = \frac{\partial l}{\partial k_1}\frac{\partial k_1}{\partial w_{21}}$$
$$= \frac{\partial l}{\partial k_1}x_2$$

41

Here is the final step. We've left out the biases: it's a good exercise to see if you can work out what the rule is for the derivative of the biases.

---

the full backpropagation algorithm



Break your function up into a computation graph.
Call the last node the *loss*.

Do a forward pass.

Start at the loss node and work your way down.

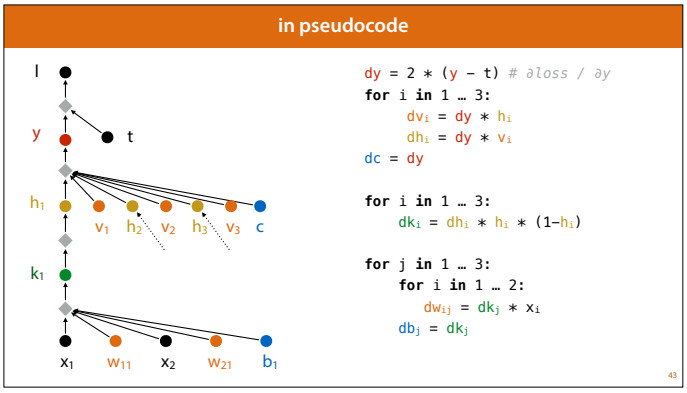
At each input to a computation, multiplying
deriv. of loss over output <- already computed
deriv. of output over input <- local deriv.
produces the deriv. of loss over input.

42

This then, is the full backpropagation algorithm. If we move down the graph from the loss node, all we need to do is compute the loss wrt the input and multiply it by the local derivative (the output over the input).

Note that a lot of this is done by hand, rather than in the program. We never actually store a computation graph in the computer, we just draw it with pen and paper and use it to work out all the rules for computing derivatives efficiently. Once we've done this, we can write these down in a computer program.

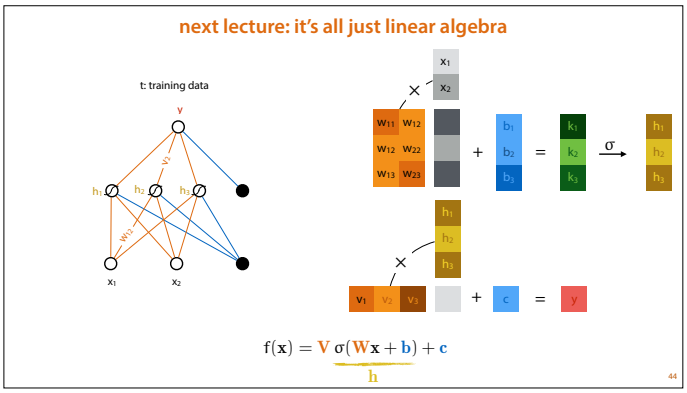
We can also have the computer do this for us: store the computation graph as an actual object in memory, and have the computer work all of this out for us. This is called automatic differentiation*, and we'll discuss it in the next lecture.*



For example, here is how backpropagation looks in pseudocode for our neural network (the diagram only shows part of the computation graph, but the algorithm is for the whole thing). In the algorithm dq is always the derivative of the loss with respect to the value q.

We start at the top, with dy. Then, we move down to the inputs of the computation that resulted in y. For each we compute their derivative by multiplying the derivative for the loss wrt the output by the local derivative of the computation.

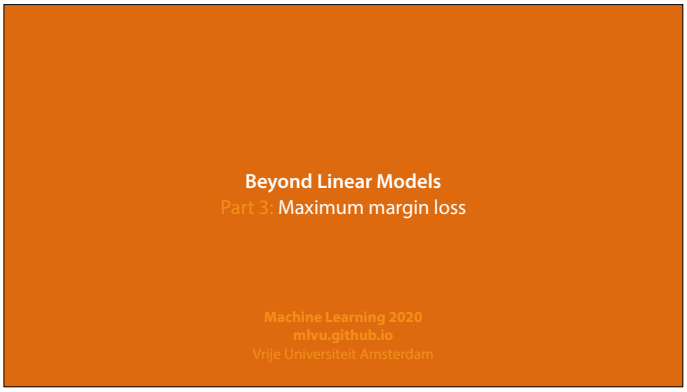
Once we've made our way down to the bottom of the graph, we've computed the derivatives for every node in the graph.



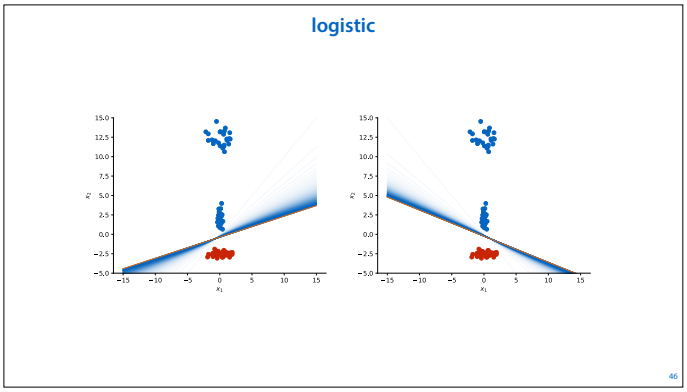
This is how backpropagation was done in the early days of neural networks (up to the turn of the century). Once we began to recognize that neural networks could actually work, we needed ways to speed up their computation. The most effective way to achieve this is to describe the whole computation in terms of matrix multiplication/addition, together with the occasional element-wise non-linear operation. This allows us to write down the operation of a neural network very elegantly, and to use highly optimized routines for matrix multiplication (possibly on special hardware like a GPU).

In order to make proper use of this, we should also work out how to do the backpropagation part in terms of matrix multiplications. That's where we'll pick up next week in the first **deep learning** lecture.

NB: This picture requires us to rename the weights in the first layer.

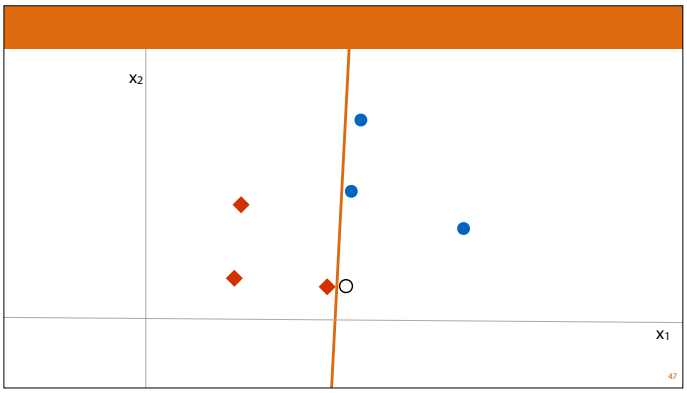


|section|Maximum margin loss|
|video|https://www.youtube.com/embed/-PvsRdlISls|



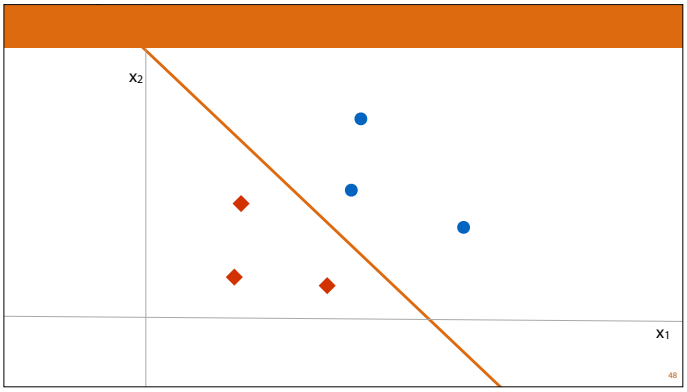
In lecture 5, we introduced the logistic regression model, with the logarithmic loss. We saw that it performed very well, but it had one problem: when the data are very well separable, it didn't have any basis to choose between two models like this: both separate the training data very well. Yet, they're very different models.

There are some tricks we can add to the logistic regression to deal with this problem, but today we'll look at a loss function that takes this problem as its starting point: the maximum margin hyperplane classifier.

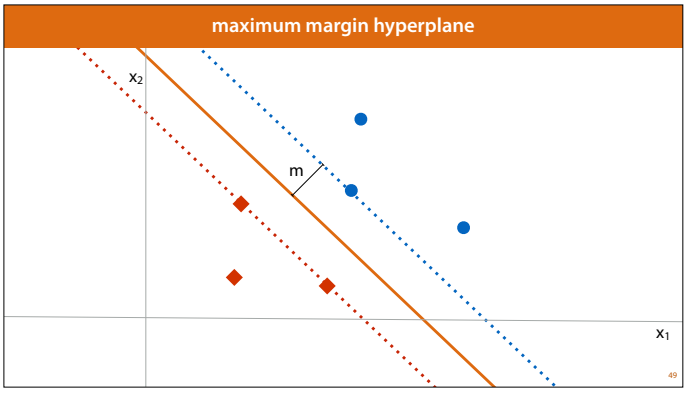


Here is an extreme example of the problem. We have two linearly separable classes and a decision boundary that separates the data perfectly. And yet, if I see a new instance that is very similar to the rightmost red diamond, but with a slightly higher x_1 value, it is suddenly classified as a blue disc.

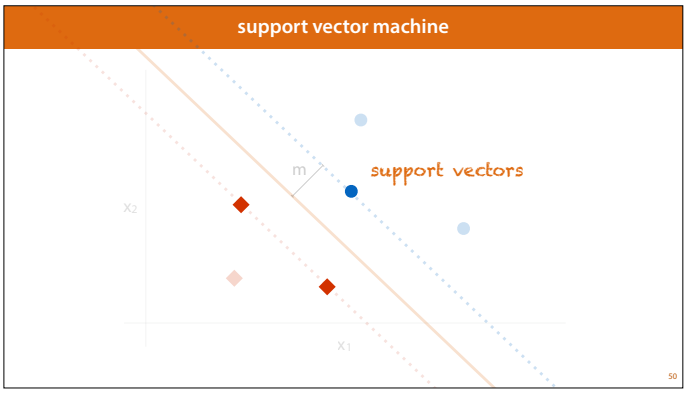
This illustrates the intuition behind the loss function we will introduce in this video. **If we see new points *near* our existing points, they should be classified the same as the existing points.** One way to accomplish this is to look at the distance from the decision boundary to the nearest red diamond and blue disc, and to *maximize* that.



What we are looking for is the hyperplane that separates the classes and has a maximal distance to the nearest positive point and nearest negative point.



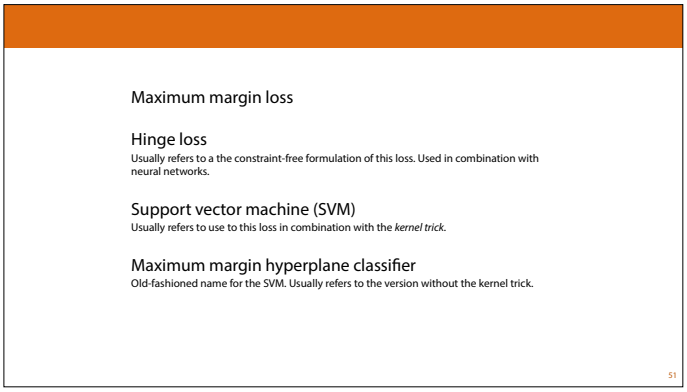
We measure the distance m at a right angle to the decision boundary. For the positive class, there is only one point nearest the margin, but for the negative class, there are two the same distance away.



The points closest to the decision boundary are called the **support vectors**. This name comes from the fact that the support vectors alone, are enough to describe the model. If I give you the support vectors, you can work out the hyperplane without seeing the rest of the data.

The distance to the support vectors is called the **margin**. We'll assume that the decision boundary is chosen so that the margin is the same on both sides.

Or, alternatively, you can imagine we are drawing parallel lines through the support vectors, and putting the decision boundary halfway between these lines.



This idea goes by many names. These all mean the same thing, but they are used in different contexts.

Given a dataset, how do we work out which hyperplane maximizes the margin?

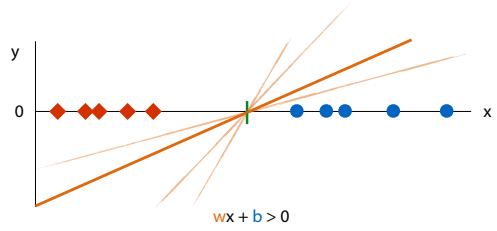
52

So, given a dataset, how do we work out which hyperplane maximizes the margin?

This is a tricky problem, because the support vectors aren't *fixed*. If we move the hyperplane around to maximize the distance to one set of support vectors, we may move too close to other points, making *them* the support vectors.

Surprisingly, there is a way to phrase the maximum margin hyperplane objective as a relatively simple optimization problem.

1D linear classifier

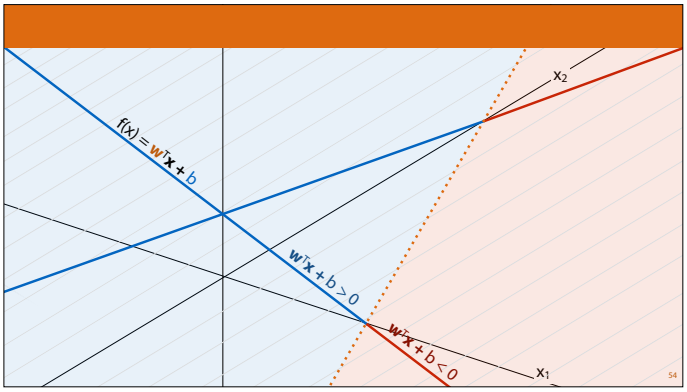


53

To work this out, let's first review how we use a hyperplane to define a linear decision boundary. Here is the 1D case. We have a single feature and we first define a linear function from the feature space to a scalar y.

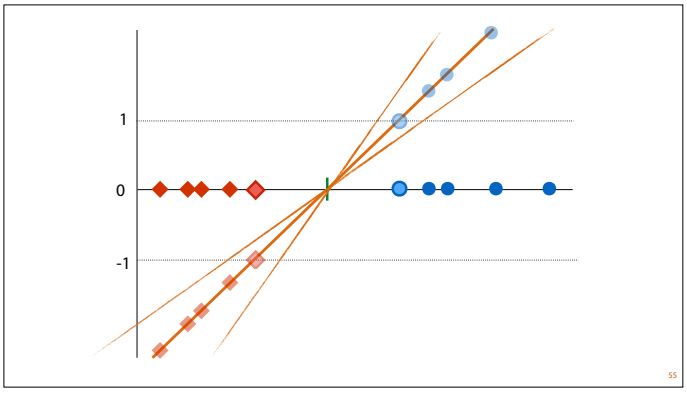
If the function is positive we assign the positive class, if it is negative, we assign the negative class. Where this function is equal to 0, where it "intersects" the feature space, is the decision boundary (which in this case is just a single point).

Note that by defining the decision boundary this way, we have given ourselves an extra degree of freedom: the same decision boundary can be defined by infinitely many hyperplanes. We'll use this extra degree to help us define a single hyperplane to optimize.



54

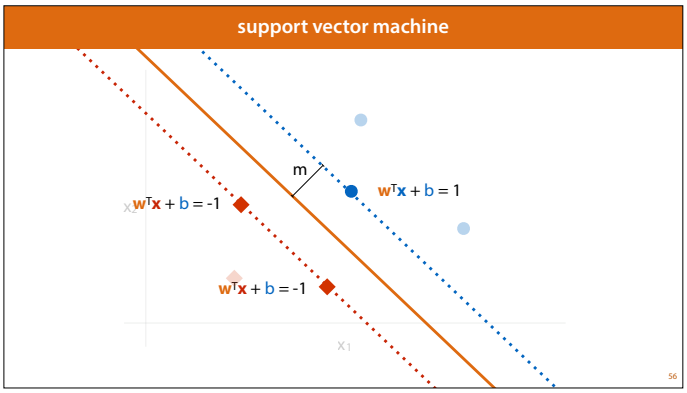
Here's the picture for a two dimensional feature space. The decision boundary is the dotted line where the hyperplane intersects the (x_1, x_2) plane. If we rotate the hyperplane about that dotted line, we get a *different* hyperplane defining *the same* decision boundary.



The hyperplane h we will choose is the one that produces y=1 for the positive support vectors and y=-1 for the negative support vectors. Or rather, we will *define* the support vectors as those points for which the line produces 1 and -1.

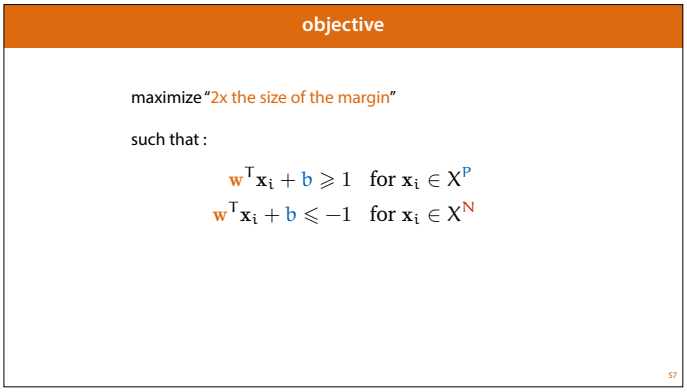
There's no guarantee that this happens at points that are in the dataset, but we will see later that this must be the case for an optimal choice of h.

For all other negative points, h should produce values below -1 and for all other positive points, h should produce values *above* 1.



This is the picture we want to end up with in 2 dimensions. The linear function evaluates to -1 for the negative support vectors, and to a lower value for all other negative points. It evaluates to 1 for the positive support vectors and to a higher value for all other positive points.

The trick we use to achieve this is to optimize **with a constraint**. We first define the margin as the distance from the decision boundary, where h evaluates to zero, to the line where h evaluates to 1, and on the other side to the line where h evaluates to -1. Then we set the constraint that all points should be on the correct side of their respective margins.

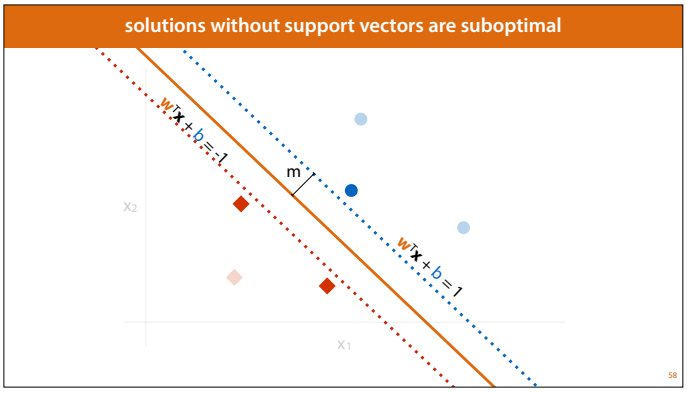


Here is our objective, written as precisely as we can manage at the moment. We will make this more precise as we move on.

The quantity that we want to maximize is "2 times the margin": the width of the band separating the negative from the positive support vectors (between the two dotted lines in the previous slide).

The constraints define the support vectors: all positive points should evaluate to 1 or higher. All negative points should evaluate to -1 or lower. Note that if we have N instances in our data, this gives us a problem with N constraints.

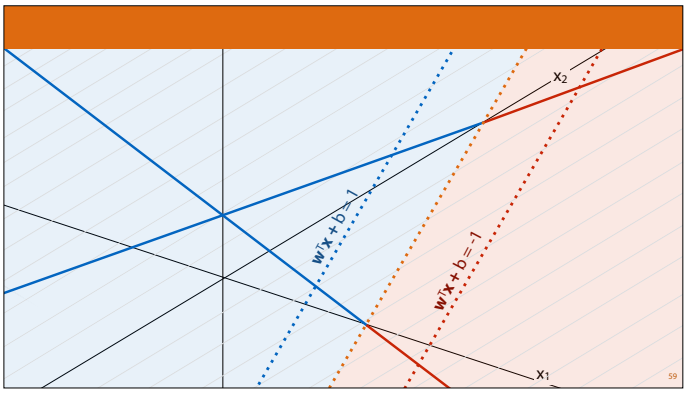
Note that this automatically ensures that the support vectors end up at -1 and 1. Why?



Here is a picture of a case where all negative points are strictly less than -1, and all positive points are strictly larger than 1. The constraints are satisfied, but there are no points on the edges of the margin: we have no support vectors.

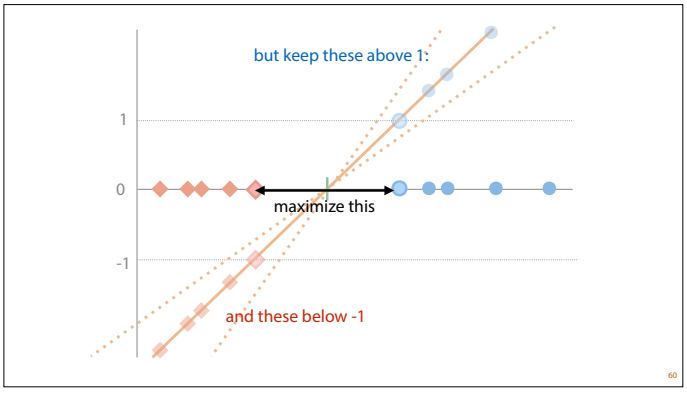
In this case, we can easily make the margin bigger, pushing it out to coincide with the nearest points. Therefore, we have not hit the maximum yet. This is not an optimal solution to our optimization problem.

Thus, any hyperplane with a maximal margin, that satisfies the constraints. must have points on the edges of its margin. These points are the support vectors.

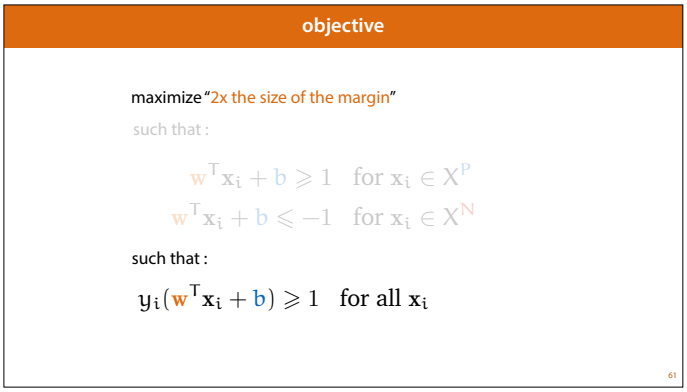


Here is the picture in 3D. Just like the hyperplane crosses the plane where y=0 to make the decision boundary, it crosses the y=1 plane to make the positive margin, and it crosses the y=-1 plane to make the negative margin.

Imagine finding a hyperplane that separates the classes, and then angling it so that he margins hit the nearest points.



Here is the picture for a single feature. We want to maximize the distance between the point where the hyperplane hits -1 and where it hits 1, while keeping the negatives below -1 and the positives above 1.



So, how do we work this into a practical optimization objective that we can actually solve?

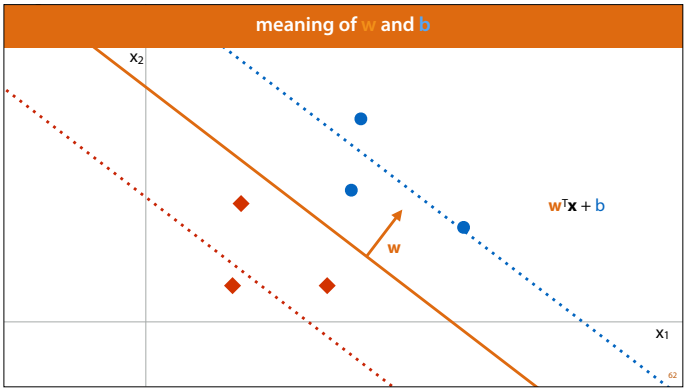
The first thing we'll do, is to simplify the two constraints for the two classes into a single constraint.

We introduce a label y_i for each point x_i which is -1 for negative points and +1 for positive points. Multiplying the left-hand side of the constraint by y_i keeps it the same for positive points and takes the negative for negative points. This means that in both case, the left hand side should now be larger than or equal to one.

This label y_i is the same label we introduced to define the least squares loss, but now we're using it in a different way. Instead of trying to map each point to its label y_i, we are fitting points to values above or below y_i.

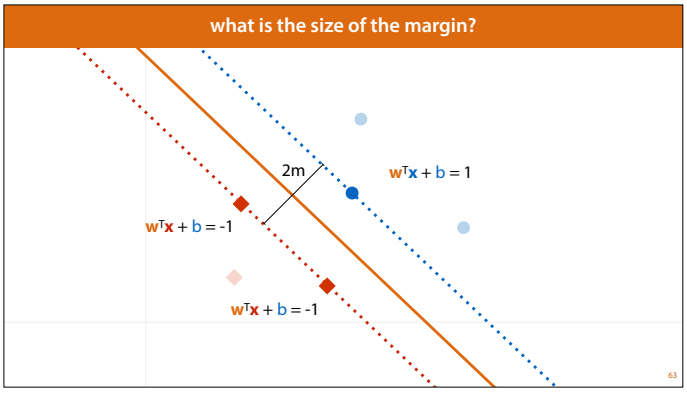
We now have a problem with the same constraint for every instance in the data.

Next, we need to make the phrase "2x the size of the margin" more precise. We know that our hyperplane, whichever hyperplane we choose, is defined by parameters **w** and b. Looking at the parameters of a particular hyperplane (good or bad), can we tell what the size of the margin is?

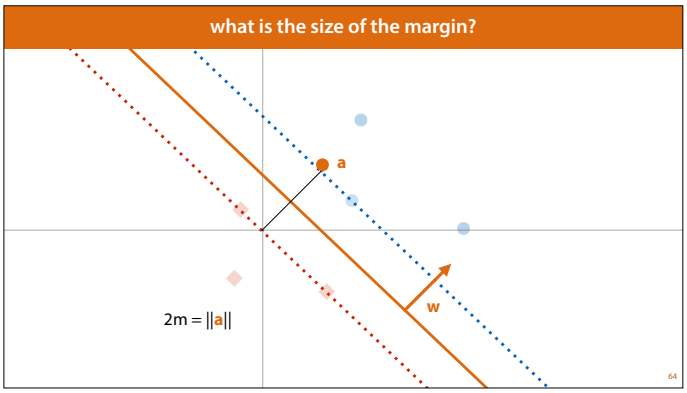


First, let's recall what the parameters mean geometrically. Remember that in the equation $\mathbf{w}^T\mathbf{x} + b$, **w** is the vector pointing orthogonally to the decision boundary. b is how high the hyperplane is at the origin.

Note that the hyperplane we have drawn here is not a solution to our problem, since it does not satisfy the constraints.



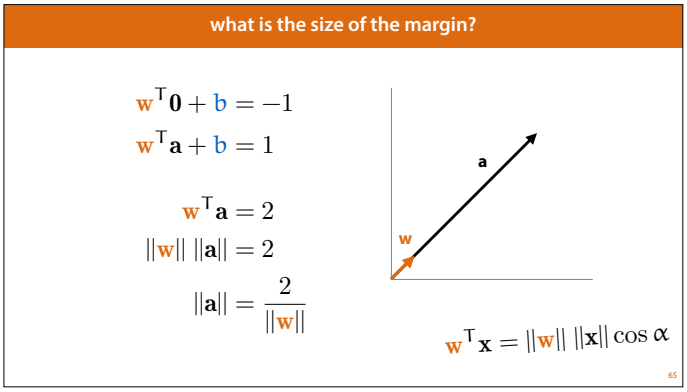
This is the value we're interested in expressing. Twice the margin.



To make the math easier, let's move the axes around so that the lower dotted line (belonging to the negative support vectors) crosses the origin. Doing this doesn't change the size of the margin.

We can now imagine a vector from the origin to the upper dotted line, at a right angle. Call this vector **a**. The length of **a** is exactly the quantity we're interested in.

Remember also that the vector **w** points in the same direction as **a**, because both are perpendicular to the decision boundary.



Because of the way we've moved the hyperplane, we know that the origin (**0**) hits the negative margin, so evaluates to -1. We also know that **a** hits the positive margin, so evaluates to +1.

Subtracting the first from the second, we find that the dot product of **a** and **w** must be equal to two.

The dot product of anything with the zero vector is always 0.

Since **a** and **w** point in the same direction ($\cos \alpha = 1$), their dot product is just the product of their magnitudes (see the geometric definition of the dot product on the right).

Re-arranging, we find that the length of **a** is 2 over that of **w**.

objective

maximize : $\frac{2}{||\mathbf{w}||}$

such that :

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geqslant 1 \quad \text{for all } \mathbf{x}_i$$

66

So, the thing we actually want to maximise is $2/||\mathbf{w}||$. This gives us a precise optimization objective.

Note that almost all the complexity of the loss is in the constraints. Without them we could just let all elements of $\mathbf{w}$ go to zero However, the constraints require the output of our model to be larger than 1 for all positive points and smaller than -1 for all negative points. This will automatically push the margin up to the support vectors, but no further.

Consider what it would mean for the elements of $\mathbf{w}$ to go to zero. $\mathbf{w}$ is the gradient of our hyperplane. It points in the direction of steepest ascent, and its magnitude indicates how steep that ascent is. The smaller $\mathbf{w}$ is, the more flat the hyperplane lies. This is what the objective says: subject to the constraint that all points are on the correct side of their respective margins, we want a hyperplane that lies as flat as possible. If you think back to the 3D picture, you should be able to imagine how this ensures as wide a margin as possible.

---

objective: hard margin SVM

minimize : $\frac{1}{2}||\mathbf{w}||$

such that :

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geqslant 1 \quad \text{for all } \mathbf{x}_i$$

67

Since we prefer to minimize instead of maximize, we take the inverse of this objective, and minimize that. The resulting classifier is called a "**hard margin**" **support vector machine** (SVM), since no points are allowed to violate the constraint and end up inside the margin.

In previous lectures we usually took the negative of the objective to turn maximization into minimization. In SVMs taking the inverse is more common, since the objective looks nicer this way.

The hard margin SVM is nice, but it doesn't work well when:

- We have data that is not linearly separable
- We could have a very nice decision boundary if we just ignored a few misclassified points. For instance, when there is a little noise, or a few outliers.

---

objective: hard margin SVM

minimize : $\frac{1}{2}\mathbf{w}^T\mathbf{w}$

such that :

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geqslant 1 \quad \text{for all } \mathbf{x}_i$$

$$||\mathbf{w}|| = \sqrt{\mathbf{w}^T\mathbf{w}}$$

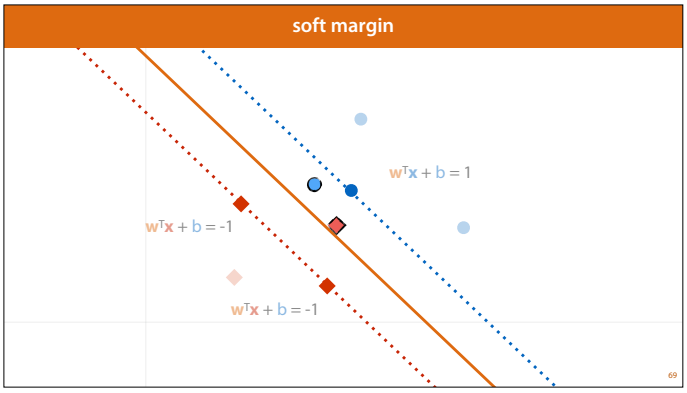
68

A common alternative is to replace the norm of $\mathbf{w}$ by the dot product of $\mathbf{w}$ with itself. This is just a question of removing the square from the norm, so it doesn't change the location of the minimum.

This is because the square is a monotonic function: if the input gets bigger, the output gets bigger.

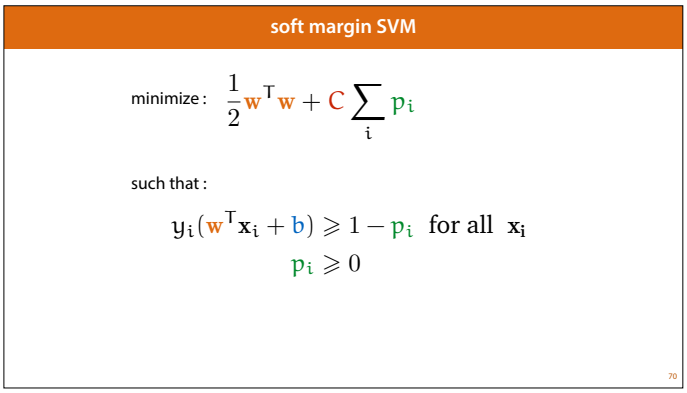
This form is easier to work with if we want to work out the gradient explicitly.

This objective is also sometimes written as $\frac{1}{2}||\mathbf{w}||^2$, which means the same thing.



To deal with such situations, we can allow a **soft margin**. In a soft margin, we allow a few points to be on the wrong side of the margin, if it helps us achieve a better fit on the rest of the points. That is, we can trade off a few violations of the constraints against a bigger margin.

These points can even be on the wrong side of the decision boundary.



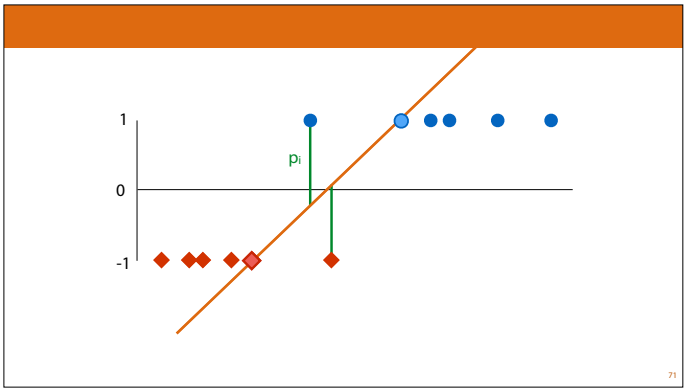
To achieve this, we introduce a **slack parameter** p_i for each point $\mathbf{x}_i$. This parameter indicates the extent to the constraint on $\mathbf{x}_i$ is *relaxed*. Our learning algorithm can set p_i to whatever it likes. If it sets p_i to zero, the constraint is the same as it was for the hard margin classifier. If it sets p_i higher than zero, the constraint is relaxed and the point $\mathbf{x}_i$ can fall inside the margin.

The price we pay is that p_i is added to our minimization objective, so the value we reach there becomes higher if we use more nonzero slack parameters.

Our search algorithm, which we will detail later, does the rest. It automatically makes the tradeoff between how much we want to violate the original constraints and how big we want the margin to be.

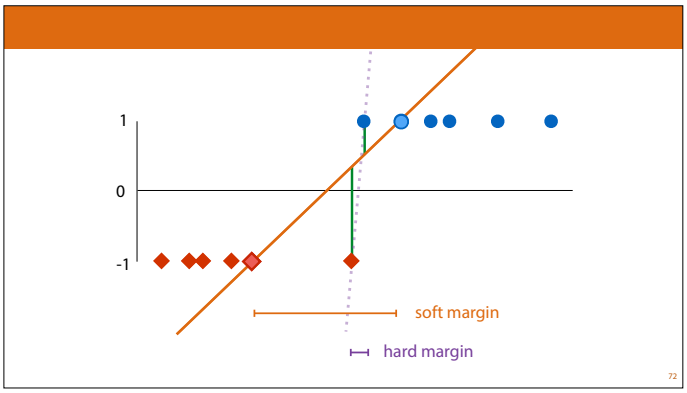
C is a hyperparameter, indicating how to balance the tradeoff.

The value of C is positive, and we usually try values on a logarithmic scale like 0.001, 0.01, 0.1, 1.0, 10 and 100. As C goes to infinity, we recover the hard margin SVM, where violating the constraints is "infinitely bad" and this never happens.



Here is what that looks like in 1D. The open points are the support vectors, and for each class, we have one point on the wrong side of the decision boundary, requiring us to pay the residual p_i as a penalty.

So, the objective function has a penalty added to it, but without this penalty, we would not have been able to satisfy the objective at all, since the two classes are not separable



However, even if the classes *are* linearly separable, it can be good to allow a little slack.

Here, the two points that would be the support vectors of the hard margin objective leave a very narrow margin. By allowing a little slack, we can get a much wider margin that provides a decision boundary that may be more likely to generalise to unseen data.

So, now that we have made our objective precise, how do we find a good solution? We haven't discussed constrained optimization much yet. It turns out, we don't necessarily *need* to use constrained optimization methods, although there is a benefit to using them. We'll look at both options.

a fork in the road

option one: express everything in terms of **w**, get rid of the constraints.

- Allows gradient descent to be used.
- Good for use with neural networks/deep learning.

this video

option two: express everything in terms of the support vectors, get rid of **w**.

- Doesn't allow error to propagate back, but …
- allows the **kernel trick** to be applied.

next video

74

The first oiption allows us to use the old familiar gradient descent, without having to worry about constraints.

The other requires us to delve into constrained optimization, which we start to do in the next video. The payoff for that is that it opens the door to the **kernel trick**.

In the rest of this video, we will work out option one.

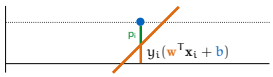
If you're in a hurry, and you just want to know the parts that are important for the course, you can skim the rest of this video and focus on the next two.

---

option one: get rid of the constraints

if $y_i(w^T x_i + b) < 1$:

$$p_i = 1 - y_i(w^T x_i + b)$$



otherwise:

$$p_i = 0$$

$1 - y_i(w^T x_i + b)$ is negative

$$p_i = \max\left(0, 1 - y_i(w^T x_i + b)\right)$$

75

To get rid of the constraints, let's look at what we know about the value of p_i.

If the constraint for $\mathbf{x}_i$ is violated, we can see that p_i makes up the difference between what $y^i(\mathbf{w}^T\mathbf{x}^i + b)$ should be (1) and what it is.

If the constraint is not violated, p_i becomes zero, and the value we computed above becomes negative.

We can summarise these two conclusions in a single definition for p_i:it is is 1 - $y^i(\mathbf{w}^T\mathbf{x}^i + b)$ if the constraint is violated and 0 otherwise. This is equal to the value max(0, 1- $y_i(\mathbf{w}^T\mathbf{x}_i + b)$) in both cases.

Since this value is always equal to p_i, we can replace p_i by it everywhere it occurs in the optimization objective

---

soft margin SVM

minimize : $\frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_i \max\left(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b)\right)$

such that : $y_i(\mathbf{w}^T\mathbf{x}_i + b) \geqslant 1 - \max\left(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b)\right)$ for all $\mathbf{x}_i$

$\max\left(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b)\right) \geqslant 0$

76

Doing this, we get a new objective function.

The new constraints are now always true. For the second one, this is easy to see, since the maximum of 0 and something is always larger than or equal to 0.

For the first, note that we worked out max(0, 1- $y_i(\mathbf{w}^T\mathbf{x}_i + b)$) as how far below 1 the value $y_i(\mathbf{w}^T\mathbf{x}_i + b)$ was. If we move it to the the other side, we get

$y_i(\mathbf{w}^T\mathbf{x}_i + b)$ + max(0, 1- $y_i(\mathbf{w}^T\mathbf{x}_i + b)$)

which must therefore be exactly equal to 1 if $y_i(\mathbf{w}^T\mathbf{x}_i + b)$ is below 1, or larger if $y_i(\mathbf{w}^T\mathbf{x}_i + b)$ is larger than 1.

Since the constraints are always true, we can remove them.

option one: unconstrained optimization

minimize :

$$\frac{1}{2}w^T w + C\sum_i \max\left(0, 1 - y_i(w^T x_i + b)\right)$$

regulariser

error

77

This gives us an **unconstrained loss function** we can directly apply to any model. For instance when training a neural network to classify, this makes a solid alternative to logarithmic loss. This is sometimes called the **L1-SVM** (loss).

There is also the L2-SVM loss where a square is applied to the p_i function, to increase the weight of outliers.

We can think of the first term as a **regularizer**. It doesn't enforce anything about how well the plane should fit the data. It just ensures that the parameters of the plane don't grow too big. We'll see more regularization in the next lecture, but this form, where we add the norm of the parameter vector to the loss function, is very common.

The highlighted part of the second term functions as a kind of error (just as we used in least squares classification, but without the square). It computes how far the model output $y_i(w^T x_i + b)$ is away from the desired value (1).

However, unlike the least squares classifier, we *only* compute this error for points that are sufficiently close to the decision boundary. For any points far from the boundary (i.e. outside the margin), we do not compute any error at all. This is achieved by cutting off any negative values. If the data is linearly separable, we could easily shrink the margin enough to make the error zero for all points, but this usually requires a w with a very high norm, so then the regulariser kicks in and start increasing so much that we prefer some points inside the margin.

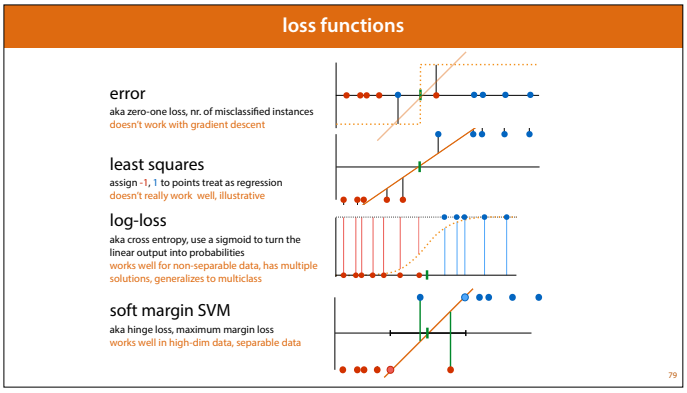
---

classification losses

- ~~Least squares loss (today)~~
- ~~Log loss / logistic regression (week 3, Probability 1)~~
- ~~SVM loss (week 3, Linear Models 2)~~

78

And with that, we have discussed our final classification loss. Let's review.



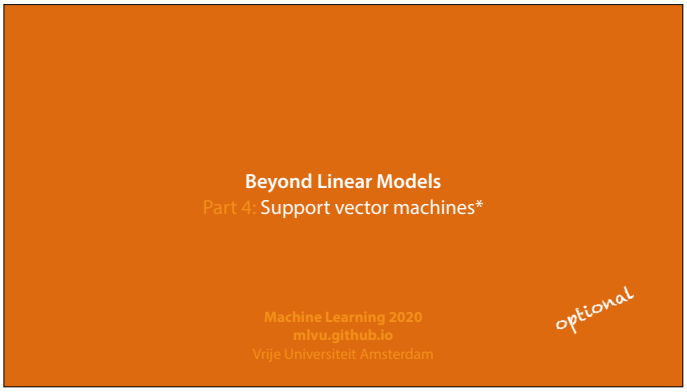
Here are all our loss functions in one handy slide.

The **error**, also known as zero one loss, is imply the number or proportion of misclasssified examples. It's usually what we're interested in, but it doesn't give us a loss surface that is suitable for searching.

The **least-squares loss** we introduced as a simple illustration of the principle of a proxy loss for the error. In practice it doesn't usually work very well, and is rarely used.

The **log loss** requires a sigmoid function to be added to the output of the linear function. It assumes that he result is the probability of the positive class applying to the instance, and it maximizes the log likelihood of the classes given the model parameters. Practically this boils down to minimizing the negative log likelihood of the correct class. This can also be derived from the cross-entropy between the true class distribution given by the data and the class distribution given by the model.

Finally, the **soft margin SVM loss**, which we've introduced today attempts to maximize the margin between the positive and negative points. It's also know as a maximum margin loss, or the **hinge loss** (since the error is fed through a maximum function, which looks like a hinge).



In this final part of the lecture, we will take a quick look at support vector machines that make use of the kernel trick. It's not exam material, so you can skip it if you need to, but we recommend at least giving a skim, so that you have a broad idea what SVMs are in case you ever encounter them in the wild.

We're skipping a large amount of technical details in this part. If you really want to know how this works all the way down to the foundations, there is an extra lecture explaining them on the website.

|section-nv|Support vector machines*
|video| |

a fork in the road

option one: express everything in terms of w, get rid of the constraints.

- Allows gradient descent to be used.
- Good for use with neural networks/deep learning.

option two: express everything in terms of the support vectors, get rid of w.

- Doesn't allow error to propagate back, but …
- allows the **kernel trick** to be applied.

81

In the previous video we introduced the maximum margin loss objective. This was a **constrained optimization** problem which we hadn't learned how to solve yet. We sidestepped that issue by rewriting it into an unconstrained optimization problem, so that we could solve it with plain gradient descent.

In this video, we will learn a relatively simple trick for attacking constrained optimization problems: the method of **Lagrange multipliers**. In the next video, we will see what happens if we apply this method to the SVM objective function.

What we'll skip:

- Lagrange multipliers
  Let you rewrite an objective with an equality constraint into one without.
- KKT multipliers/conditions
  Let you rewrite an objective with inequality constraints into one without.
- Details of the kernel trick

82

To avoid overloading this lecture, we will skip a lot of the technical details. We will give you a very high level view of what support vector machine can do. This should help you recognize what they are, and give you a sense of how they work, in case you ever need to apply them.

If you ever need to understand them properly, all the technical details are available on the website in an extra lecture.

$$\text{minimize } \frac{1}{2} w^T w + C \sum_i p_i$$

$$\text{such that } y_i(w^T x_i + b) - 1 + p_i \geqslant 0$$

$$p_i \geqslant 0$$

KKT/Lagrangian magic

$$\text{minimize } \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x_i^T x_j} - \sum_i \alpha_i$$

$$\text{such that } 0 \leqslant \alpha_i \leqslant C$$

$$\sum_i \alpha_i y_i = 0$$

83

The key idea behind all this technical stuff is that we can rewrite our *maximum margin objective* from the previous slide into its *dual form*. This is what the KKT method does for us, it gives us a new minimization objective where some of the variables have disappeared. Under the right conditions, both objectives have the same solution.

In return we also get some extra variables α_i. You can think of these as weights, we get one for every instance $\mathbf{x}_i$, y_i in our data. The hyperplane parameters have disappeared and these αs are the only parameters of our problem now.

There isn't much more intuition we can give you. Without a whole lot of math, you'll just have to take our word for it that these two problems are equivalent. We can find the optimal maxim-margin hyperplane loss using the problem at the top or the problem at the bottom.

The reason this dual problem is so interesting is that the only thing we ever need to compute on our data **is the dot product between every pair of instances in our data**. That is, we don't need the original features: if I give you all

dot products between all pairs of instances in a dataset, you can fit an SVM to it without every seeing anything else of the data.

This is the idea that leads to the **kernel trick**.

---

the kernel trick

If you have an algorithm which operates only on the dot products of instances, you can substitute the dot product for a kernel function.

84

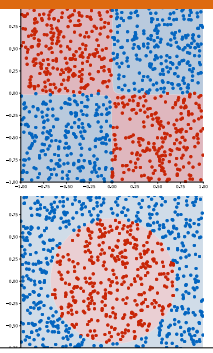
What if I didn't give you the actual dot products, but instead gave you a different matrix of values, that *behaved* like a matrix of dot products.

The idea is that it's possible to design a high-dimensional feature space in such a way that that you can very cheaply compute dot products without ever having to compute the high dimensions vectors.

---

making linear models more powerful

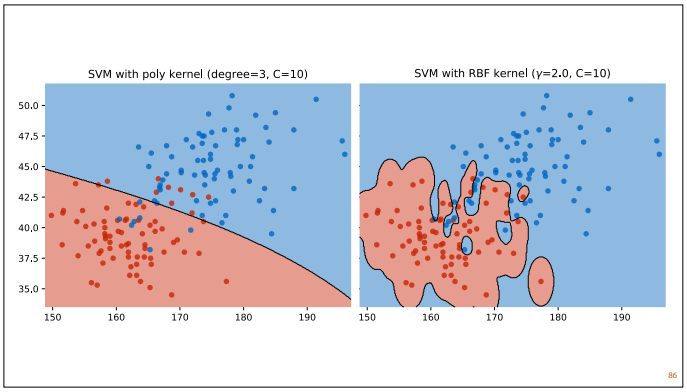
d	p	d * p
0.75	0.98	0.74
-0.66	-0.32	0.21
-0.45	0.84	-0.38
0.93	0.78	0.72
-0.42	0.24	-0.10
-0.02	0.43	-0.01
-0.74	0.58	-0.43
-0.41	-0.41	0.17
0.59	0.72	0.42



85

Remember, by adding features that are derived from the original features, we can make linear models more powerful. If the number of features we add grows very quickly (like if we add all 5-way cross products), this can becomes a little expensive (both memory and time wise).

The kernel trick is basically a souped-up version of this idea. We expand our features into a high dimensional space, except we never actually compute the feature expansion. We just compute the dot products in the expanded feature space directly.



Here's a plot for the dataset from the first lecture. As you can tell, the RBF kernel massively overfits for these hyperparameters, but it does give us a very nonlinear fit.

---

extending the feature space

x_1	x_2	x_1^2	x_1x_2	x_2^2	
3	105	9	315	11025	male
1	110	1	110	12100	male
7	119	49	833	14161	male
8	120	64	960	14400	male
9	120	81	1080	14400	male
12	119	144	1428	14161	female
8	122	64	976	14884	female
8	125	64	1000	15625	female
9	125	81	1125	15625	female
9	132	81	1188	17424	male
14	128	196	1792	16384	female

87

Let's look at an example. The simplest way we saw to extend the feature space was to add **all cross-products**. This turns a 2D dataset into a 5D dataset. It also more than doubles the amount of data we need to store our dataset, and the amount of time required to, for instance, compute a dot product in this space.

This is not usually a bottleneck, but we want to expand this idea to thousands or even millions of extra features.

Again, to fit an SVM, all we need is the dot products between pairs of instances in this 5D space. Let's see if we can compute those, or something similar, without explicitly computing the 5D vectors.

$$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

$$k(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T\mathbf{b})^2$$

88

Here are two 2D feature vectors. What if, instead of computing their dot product, we computed the *square* of their dot product.

It turns out that this is equal to the dot product of two other *3*D vectors **a'** and **b'**.

$$(\mathbf{a}^T\mathbf{b})^2 = (a_1b_1 + a_2b_2)^2$$

$$= {a_1}^2{b_1}^2 + 2a_1b_1a_2b_2 + {a_2}^2{b_2}^2$$

$$= {a_1}^2 \cdot {b_1}^2 + \sqrt{2}a_1a_2 \cdot \sqrt{2}b_1b_2 + {a_2}^2 \cdot {b_2}^2$$

$$= \begin{pmatrix} {a_1}^2 \\ \sqrt{2}a_1a_2 \\ {a_2}^2 \end{pmatrix}^T \begin{pmatrix} {b_1}^2 \\ \sqrt{2}b_1b_2 \\ {b_2}^2 \end{pmatrix}$$

89

The square of the dot product in the 2D feature space, is equivalent to the regular dot product in a 3D feature space. The new features in this 3D space can all be derived from the original features. They're the three cross products, with a small multiplier on the a_1a_2 cross product.

$$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

$$k(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T\mathbf{b})^2$$

$$\mathbf{a'}^T\mathbf{b'} = (\mathbf{a}^T\mathbf{b})^2 \qquad \mathbf{a'} = \begin{pmatrix} {a_1}^2 \\ {a_2}^2 \\ \sqrt{2}a_1a_2 \end{pmatrix}, \mathbf{b'} = \begin{pmatrix} {b_1}^2 \\ {b_2}^2 \\ \sqrt{2}b_1b_2 \end{pmatrix}$$

90

That is, this kernel function k doesn't compute the dot product between two instances, but it does compute the dot product in a feature space of *expanded* features. We could do this already, but before we had to actually *compute* the new features. **Now, all we have to do is compute the dot product in the original feature space and square it.**

$$\text{minimize} \quad \frac{1}{2}\sum_i \sum_j \alpha_i\alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j) - \sum_i \alpha_i$$

$$\text{such that} \quad 0 \leqslant \alpha_i \leqslant C$$

$$\sum_i \alpha_i y_i = 0$$

91

Since the dual solution to the SVM is expressed purely in terms of the dot product, we can replace the dot product this kernel function. We are now fitting a line in a higher-dimensional space, without computing any extra features explicitly.

Note that this only works because we rewrote the optimization objective to get rid of **w** and b. Since **w** and b have the same dimensionality as the features, keeping them in means using explicit features.

Saving the trouble of computing a few extra features may not sound like a big saving, but by choosing our kernel function cleverly we can push things a lot further.

a kernel function

$$k(\mathbf{x_i}, \mathbf{x_j})$$

A function that computes the dot product of x_i and x_j in a *different* feature space, without explicitly computing those features.

92

For some expansions to a higher feature space, we can compute the dot product between two vectors, **without explicitly expanding the features**. This is called a **kernel function**.

There are many functions that compute the dot product of two vectors in a highly expanded feature space, but don't actually require you to expand the features.

There are some straightforward conditions for when a given function of two vectors is a kernel. We won't worry about that now, and just look at some commonly used kernels, assuming that others have done the work to show that these actually are kernels.

---

polynomial kernel

$$k(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T\mathbf{b} + 1)^d$$

feature space for d=2: all squares, all cross products, all single features

feature space for d=3: all cubes and squares, all two-way and three-way cross products, all single features.

This is already a **big** feature space.

93

Taking just the square of the dot product, as we did in our example, we lose the original features. If we take the square of the dot product **plus one**, it turns out that we retain the original features, *and* get all cross products.

You'll show how this works in the homework.

If we increase the exponent to d we get all d-way cross products. Here we can see the benefit of the kernel trick. Imagine setting d=10 for a dataset with a modest 10 features. Expanding all 10-way cross-products of all features would give each instance *10 trillion* expanded features. We wouldn't even be able to fit one instance into memory.

However, if we use the kernel trick, all we need to do is to compute the dot product in the original feature space, add a 1, and raise it to the power of 10.

d is a hyperparameter: increasing it does not make the algorithm much more expensive, but it does increase your (implicit) feature space so much that you risk overfitting, so

---

RBF kernel

$$k(\mathbf{a}, \mathbf{b}) = \exp\left(-\gamma\|\mathbf{a} - \mathbf{b}\|\right)$$

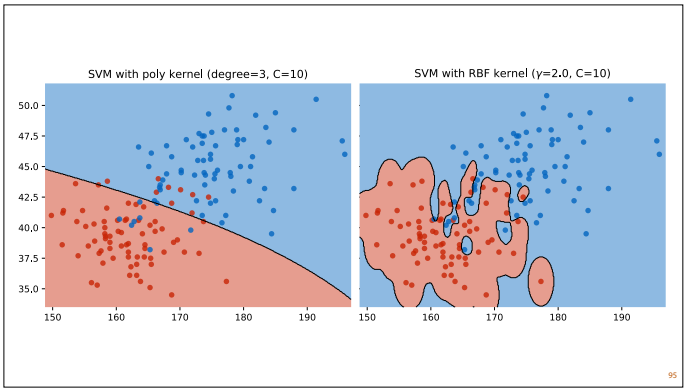
feature space: infinite-dimensional

94

If ten trillion expanded features sounded like a lot, here is a kernel that corresponds to an infinite-dimensional expanded feature space. We can only approximate this kernel with a finite number of expanded features, getting closer as we add more. Nevertheless, the kernel function itself is very simple to compute.

Gamma is another hyperparameter.

Because this is such a powerful kernel, it is prone to overfitting.



Here's a plot of the SVM decision boundary with a poly kernel and an RBF kernel, on some simple dataset.

As you can tell, the RBF kernel massively overfits for these hyperparameters, but it does give us a very *nonlinear* fit.

---

kernels in data space

text, **DNA**, **proteins**: string kernels (inspired by edit distance)

graphs: Weisfeiler-Lehman distance

96

One of the most interesting application areas of kernel methods is places where you can turn a distance metric in your data space directly into a kernel, **without first extracting any features at all**.

For instance for an email classifier, you don't need to extract word frequencies, as we've done so far, you can just design a kernel that operates directly on strings (usually based on the edit distance).Put simply, the fewer operations we need to turn one email into another, the closer we put them together. If you make sure that such a function behaves like a dot product, you can stick it in the SVM optimizer as a kernel. **You never need to deal with any features at all.** Just the raw data, and their dot products in some feature space that you never compute.

This approach has often been used to analyze DNA and protein sequences in bioinformatics.

If you're classifying graphs, there are distance metrics like the Weisfeiler-Lehman algorithm that you can use to define kernels.

---

using kernel SVMs

Normalize your data.

Pick a kernel (`linear`, `rbf`, `poly`)

Pick a C and the hyperparameters for your kernel (d, γ)

```
In [106]: from sklearn.svm import SVC

          lin = SVC(kernel='rbf', gamma=0.1, C=10)
          lin.fit(x, y)
```

97

Kernel SVMs are complicated beasts to understand, but they're easy to use with the right libraries. Here's a simple recipe for fitting an SVM with an RBF kernel in sklearn.

why did neural networks come back?

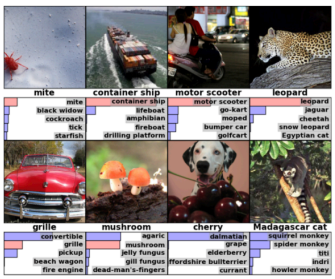
Quadratic vs. linear training time.
SVM training needs to see all pairs of instances: $O(N^2)$
Neural net training needs k passes over the data: $O(kN)$

Good SVM performance required hand-designed kernels.

Deep neural nets matured, and hardware caught up with them.
LSTMs and ConvNets, both invented in 1998, provided the first breakthroughs.

Machine learning culture changed: empirical evidence of performance became acceptable without proof of convergence/learnability.

Neural nets require a lot of passes over the data, so it takes a big dataset before kN becomes smaller than N^2, but eventually, we got there. At that point, it became more efficient to train models by gradient descent, and the kernel trick lost its luster.



mlcourse@peterbloem.nl

And when neural networks did come back, they caused a revolution. That's where we'll pick things up next lecture.